Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 359 – 368

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

A Methodology for Invasive Programming on Virtualizable Embedded MPSoC Architectures

Alexander Biedermann, Sorin A. Huss

Integrated Circuits and Systems Lab, TU Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany

Abstract

Exploiting the huge logic resources in current embedded devices has led to a plethora of on-chip multi-processor architectures. However, besides instantiating more and more soft-core processors on a chip, developing applications suited for such architectures still remains a hard task. A further step in the evolution of embedded multi-processing might be the so called Invasive Programming. In this paradigm, an application may be switched from sequential to parallel execution at runtime. A task may then dynamically invade currently unused processor resources in a multi-processor system to resume in parallel execution mode. This hardens existing problems, however, because not only the development of suited software, but also the creation of multi-processor architectures supporting this paradigm is needed. Therefore, this work presents a concise methodology to enable Invasive Programming properties on an embedded Multi-Processor System-on-Chip (MPSoC). This is achieved by combining a designer-guided code parallelization approach with a virtualizable, generic, and scalable embedded MPSoC architecture. To resolve data dependencies during task invasion, a processor-independent task-based communication scheme for the MPSoC is proposed. Moreover, a tool framework dedicated to the generic creation of virtualizable MPSoC is provided. The approach is demonstrated by the generation of an MPSoC featuring eight processors executing an application which dynamically switches at runtime between sequential and parallel execution.

Keywords: FPGA, MPSoC, Virtualization;

1. Introduction

In past years, research has spent much effort in efficiently exploiting the steadily increasing resources in digital devices. In personal computing, at some point, the focus faded from building more and more complex processor architectures towards instantiating the same processor multiple times on a chip. In the area of embedded computing, the trend of applying multi-core/multi-processor paradigms was adopted with a certain delay. Nowadays, a plethora of different multi-core and multi-processor architectures for embedded computing exists. However, most of the legacy software code used in embedded computing is sequential. To overcome this limitation, paradigms such as OpenMP [1] feature program statements to manually express parallelization. These statements are later compiled into parallelized sections. Approaches such as CUDA and OpenCL that originally were intended for graphics cards were recently adopted for FPGAs, see [2], [3]. Assuming the existence of smooth approaches for code parallelization, embedded devices will often face a multitude of environmental and inner effects to which

*Corresponding author. Tel.: +49-6151-166710; fax: +49-6151-166710.
E-mail address: biedermann@iss.tu-darmstadt.de.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.199

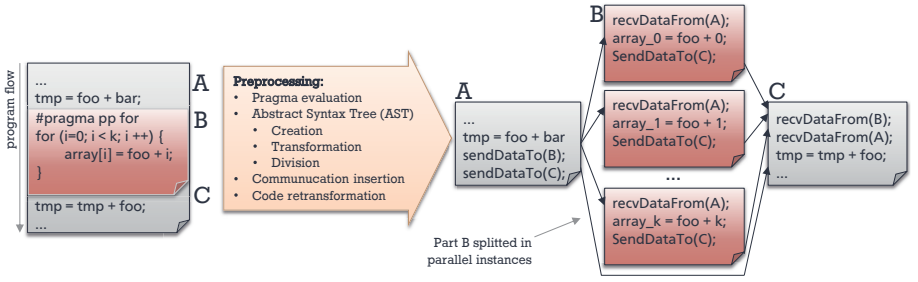

Fig. 1. Splitting of sequential Software into a Data-Flow Structure featuring a parallel Execution Part (Rectangles with flipped corners denote program code).

they have to cope with such as fluctuations in power supply, the need to preserve energy when being driven by battery, and defects of parts of the hardware despite having to continue operation reliably just to name a few. For those scenarios, several actions are required. The system might disable several tasks and decrease the number of active processors to preserve energy. Instead of completely disabling a parallelized task, the system might dynamically decrease its degree of parallelism and, thus, reduce the number of allocated processors. This paradigm of switching between sequential execution and different levels of parallelization during runtime is known as *Invasive Programming*, since unused processors in the system may be *invaded*. In contrast to common multi-threaded multi-processor architectures, invasive architectures feature a high degree of dynamism regarding task-processor bindings and execution modes of tasks. Moreover, the work presented in the scope of this paper completely skips the usage of an underlying operating system or kernel for tasks, thus saving overhead both in execution time and memory. In [4], several requirements towards Invasive Programmable MPSoC are postulated. Programming languages have to be aware of the underlying architecture that provides processing elements and communication mechanisms. The mapping of computations may not be static in order to allow for adaptability. And, at last, compilers as well as simulators have to support invasive programs and architectures. In this context, an approach is needed to compile code with different granularities of parallelization as well as a dedicated multi-processor architecture that supports smooth task migration and dynamic mapping of tasks. To allow for debugging, simulators should support such systems as well.

In this work we introduce a methodology which covers all the requirements mentioned before. At first, in Section 2, we will expand upon a code parallelization approach that preprocesses pragmas inserted into sequential code to generate parallel structures. Second, in Section 3, we exploit an existing virtualizable multi-processor system-on-chip (MPSoC) architecture that features smooth task migration and processor resource sharing. By generalizing the virtualization concept of this architecture, it now becomes possible to transparently switch between sequential and parallel implementations of the given software. For resolving dynamic data dependencies between parallelized tasks, a task-based processor-independent communication system is presented in Section 3.2.2. The design process resulting from this methodology is guided by a dedicated design framework developed for this purpose, cf. Section 4. To demonstrate the feasibility of our methodology, in Section 5, both a sequential and a set of several parallel representations of a downscaling image filter are generated and mapped to an MPSoC featuring up to eight soft-core processors on a Virtex-6 FPGA. Thus, a transition between sequential and several parallel execution modes at runtime is achieved by allocating or de-allocating processor resources via the virtualization scheme and by dynamically distributing data dependencies between a varying number of parallelized program parts. Therefore, the proposed Invasive Programming methodology combines a programming solution for parallelizing software with a hardware architecture that enables the dynamic transition between sequential and parallel execution mode.

2. Code Parallelization of Embedded C Code

The parallelization of sequential software is a common problem when facing multi-processor architectures. The reconfigurability of FPGAs offers a huge variability in terms of multi-processor computing. The number of soft-core processors employed that way may exactly be tailored to the needs of parallelized software. We expand

upon a code parallelization approach presented in [5] which matches parallelized software to an automatically generated, suited multi-processor architecture. Before highlighting the improvements made to this approach in order to support Invasive Programming, we give a short overview about the basic parallelization concept.

2.1. Code Parallelization by Pragma Insertion

In [5] the starting point is sequential (legacy) software code. The approach features dedicated precompiler directives, called "pragmas" that are manually inserted into the code to denote parallelism. Tools such as Pareon [6] exist which identify sections in the code suited for parallelization, i.e., for a pragma placement. One of the most apparent code sections to look for intrinsic parallelism is loops. Within a pragma the designer may further define `maxnum`, a parameter denoting an upper bound for the desired degree of parallelization. Pragmas will then be interpreted by a precompiler, which tries to parallelize the section marked by the pragma. Other projects, such as OpenMP, make also use of pragmas [1]. In OpenMP, however, generated parallelism focuses mainly on multi-threaded multi-core systems, while this approach mainly aims at multi-processor architectures based on soft-core processors on FPGAs. For Invasive Programming schemes with OpenMP, the work of [7] features iOMP, an extension of the OpenMP application programming interface to create resource-aware parallel tasks in the area of high performance computing on CPUs. Again, this method cannot directly be exploited for MPSoCs on FPGAs.

When compiling software featuring pragmas, a precompiler builds an abstract syntax tree (AST) representation of the code. For each pragma evaluation, the respective AST is transformed and separated into three parts as visible from Figure 1. Parts A and C of the AST contain the code which will be executed before or after the section marked by a pragma. Part B contains the code enclosed by the pragma. This is the code section to be instantiated multiple times after the transformation.

Splitting the AST into several parts causes data dependencies. So, static dependencies are copied to each of the corresponding parts. In order to resolve dynamic dependencies at runtime, however, communication has to be defined between disjoint AST parts. Data causing dynamic dependencies may then be exchanged during runtime. Therefore, in the re-transformation process from AST representations to code, point-to-point connections supported by the target architecture are automatically inserted into the code. After AST manipulation and re-transformation into code, as a result, a data flow-like structure is generated. Data stemming from dynamic dependencies is forwarded from part A over part B to part C with no backward edges as depicted in the right hand side of Figure 1. Some dependencies, such as the variable "tmp" in Figure 1 which only exist between parts A and C, are resolved by establishing a direct connection from A to C without traversing the B parts. While this approach is very well-suited for data-flow oriented applications, the designer may face a significant performance drop in applications featuring many memory accesses. In this case, the transfer of data between split tasks may pose a bottleneck. Therefore, this approach is being expanded in the sequel in order to support Invasive Programming on a generic MPSoC architecture.

2.2. Dedicated Enhancements for Invasive Programming Paradigm

In order to be able to switch between sequential and several parallel execution modes during runtime, it is necessary to modify the code parallelization method detailed in the previous section. At first, the generation of a set of implementations with various degrees of parallelization is automated by repeatedly executing the pragma evaluation while altering the `maxnum` (cf. Section 2.1) parameter. Alternatively, by placing pragmas at other suited code sections, further implementation variants may be generated as well.

Now, the fundamental problem of Invasive Programming of defining the points where it is possible to switch between sequential and parallel execution modes comes up. A solution supporting any point in time for switching within the execution flow would be most preferable. However, this will result in a considerable overhead caused by the code structures needed for data forking and re-aggregation. Therefore, we advocate to define distinct *switching points* (SP) for transitions between sequential and parallel execution modes. While the virtualization hardware architecture highlighted in Section 3 supports the interruption and migration of tasks at an arbitrary point in time, the change of execution modes is restricted to happen only at such switchting points. For example, the junction between code sections A and B in the sequential representation depicted in Figure 1, may serve as a SP.

After pragma evaluation, the A part (cf. Figure 1, right hand side) of a parallelized application contains code for the resolution of data dependencies, i.e., to send commands. This portion is copied into the sequential

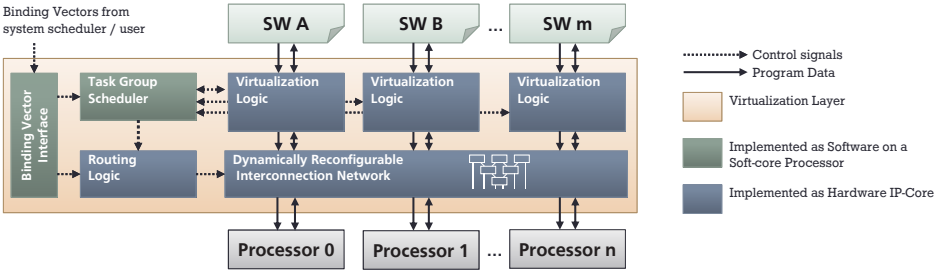

Fig. 2. Basic Architecture of a Virtualizable MPSoC.

implementation at the position of the SP. The A part of the implementation generated by pragma evaluation is then discarded. Next, a small portion of code is wrapped around this dependency resolution code. This wrapper reads the desired execution mode from an external trigger. In case that a parallel execution is requested, then the instructions for dependency resolution are executed. Otherwise, the dependency resolution commands are skipped and the program continues to execute in sequential mode. To include further parallel implementations of the application, two alternatives exist: On the one hand, the code for dependency resolution may be generalized so that it supports sending data to an arbitrary number of B parts. On the other hand, for each parallel implementation to be integrated its specific dependency resolution code may be copied into the sequential implementation and the code selecting the execution mode has to be slightly adapted. This alternative, however, leads to an increased size of the program memory. In future work, all these steps will be fully automated.

We now discuss how to map this structure onto a generic, virtualizable MPSoC architecture. The concept of the underlying MPSoC and the enhancements dedicated for the support of the Invasive Programming scheme are then highlighted in the following section.

3. Virtualizable MPSoC Architecture

3.1. Architecture Overview

In [8], a virtualization approach for a set of embedded soft-core processors was presented. In this architecture, the execution of completely independent software tasks may be shifted transparently between processor set elements during runtime. This is achieved without an underlying operating system or kernel solely by means of a code injection approach. Therefore, the instruction and data memory interfaces of each processor are routed through a virtualization unit. This unit may disconnect the memories of a processor and subsequently inserts dedicated portions of machine code that cause this processor to output its context, such as register contents. This context may then be fed into another processor in the system that is detached in a similar way. Execution of tasks is then seamlessly resumed by re-connecting the memories to the processors. A related switching procedure with context extraction, however dedicated to the migration of hardware tasks, was presented in the work of [9]. In contrast, other approaches, such as in [10], exploit micro kernels to provide virtualization, e.g., on an ARM core. The approach in [8] avoids safety concerns arising from malicious address space violations of tasks by keeping the memory spaces of independent software tasks completely disjoint at any time. However, this work was limited to a set of three processors only and does not feature a sound methodology to define or to update processor-task bindings and task scheduling. In contrast, the work in [11] does provide techniques for task scheduling by combining scheduling actions performed either in application or kernel mode, thus providing both a fair processor access and maintaining real-time properties despite scheduling events. Since scheduling techniques were missing in [8], the work in [12] expands the original approach to feature scheduling of an arbitrary number of software tasks on a user defined number of processors. Transparent sharing of a processor resource between tasks, scheduling of task groups, and dynamic processor (re-)allocation are intrinsic properties enabled by the employed dynamically reconfigurable permutation network connecting tasks and processors. In this approach, at any time, new processor-task bindings and task groups sharing a processing resource may be made available to the system. At such update events, a virtualization procedure halts the execution of tasks. A dynamic reconfiguration of the

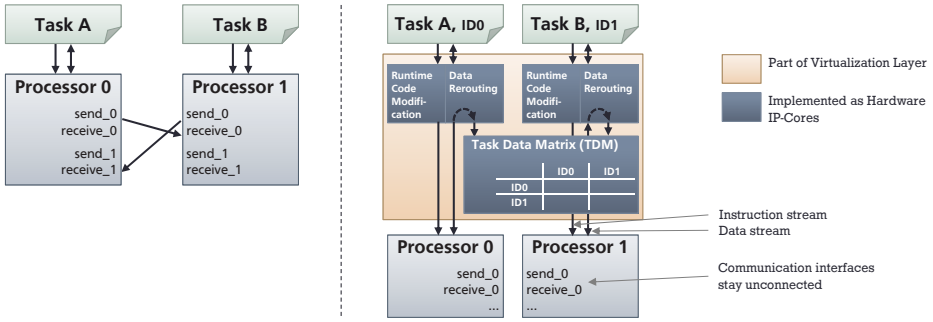

Fig. 3. Fixed Communication Links in a Static Multi-Processor System (left hand side) vs. Processor-independent Communication by Runtime Code Modification (right hand side).

permutation network follows and sets up the new processor-task bindings. According to [12], a new configuration of this network is found and applied in less than 155 clock cycles. This is due to the fast sorting-like routing algorithm which may be applied on the employed Max-Min-Network. Task execution is then seamlessly resumed with updated bindings. The overall procedure takes a maximum of 155 clock cycles, depending on whether the complete software-processor binding or only a part of it is updated. This is significantly faster than the time needed for a simple task-switch in multi-threaded kernels on the same processor type. Measurements have shown that the Xilkernel, a kernel for the (single-core) MicroBlaze processor provided by the vendor Xilinx, takes approximately 1,350 clock cycles for a thread switch. Therefore, the advocated multi-processor architecture is able to switch task contexts about ten times faster compared to common kernels for embedded processors. The structure of this architecture is depicted in Figure 2. Here, neither the processors nor the tasks need any information about whether a task is a sole user of a processor or whether several tasks share the same processor resource in a time multiplexing scheme. While the works in [8] and [12] introduced a hardware architecture which enables task virtualization, this contribution presents a methodology that exploits and extends the features provided by the previously published generic virtualizable hardware architecture.

In order to support invasive features, the architecture has to provide a task communication scheme to resolve data dependencies between parallelized tasks as detailed in Section 2.2. However, a communication concept between tasks is missing in [12]. This is caused by the fact that tasks have no knowledge about on which processor their potential communication partner is currently being executed. Therefore, this work proposes a processor-independent task-based communication system for the virtualizable MPSoC.

3.2. Task-based Communication Scheme

3.2.1. Motivation for Processor-Independent Communication

Data exchange in a multi-processor architecture is being addressed by several approaches. A shared memory space which may be accessed by all processors might be exploited for data transfer. Usually, in shared memory, a dedicated portion of the available memory space is dedicated to a specific processor. Since the execution of tasks may be moved to other processors of the system during runtime by the presented approach, an assignment of memory regions to tasks but not to processors would be desirable. However, the architecture of FPGAs is not well-suited to establish fast shared memories. Onboard memory primitives, the BlockRAM, are limited in size and, furthermore, restricted to two read/write ports only. In contrast, appropriate large multi-port memories often lead to inefficient synthesis results for FPGAs. Using an off-chip memory as an alterantive, however, reduces the access speed and, thus, the data throughput due to a higher latency.

Another possible communication infrastructure is a bus-based architecture. As every task is eligible to send data, two alternatives for bus-based communication exist: First, a multi-master bus may be established. Here, at any point in time, just one bus participant, i.e., one task, may access the bus to write data. A central bus arbiter instance is therefore needed to control write access to the bus. Only one write access at any point in time and a centralized arbiter, however, may result in significant bottlenecks. Second, for each task in the system a single-master bus can be instantiated. Each task may then write exclusively to its bus. However, this will generate a

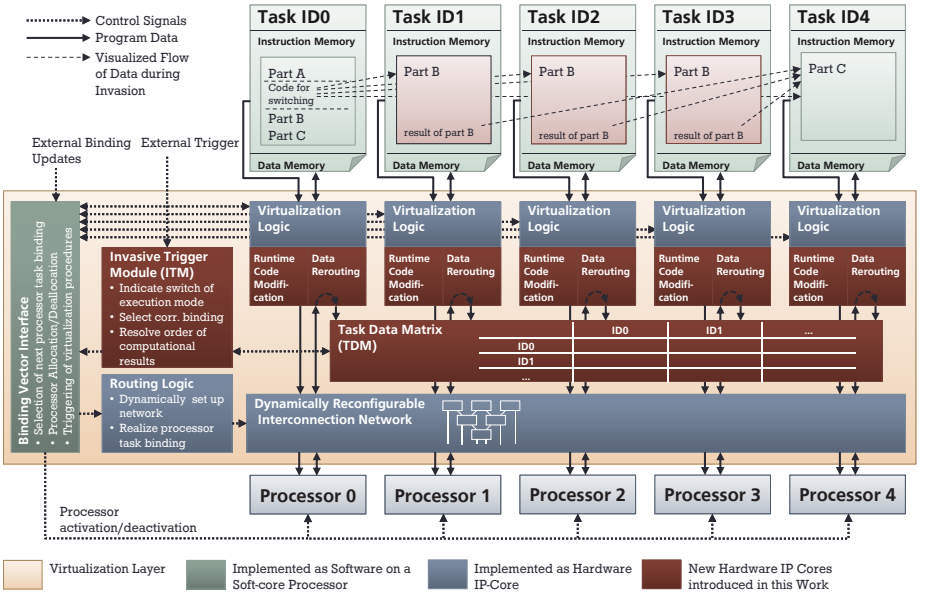

Fig. 4. Expanded Virtualization Layer and Code Distribution on an MPSoC featuring five Processors and a Parallelization Degree of three.

resource overhead as well as an increase in routing complexity, since each of these buses has to be routed to each and every processor in the system.

Therefore, the most suited approach for data exchange in the proposed virtualization architecture is in our view a modified point-to-point communication. Point-to-point connections represent usually the fastest communication scheme with moderate resource consumption, they are normally established by static links between processors. This scheme is highlighted in Figure 3, left hand side. Since the binding is static in conventional systems, a designer knows on which processor any task is being executed. Thus, to send data to another task, a task instance may address the corresponding communication interface of its processor connected to the processor executing the receiving task. In the virtualizable MPSoC highlighted in the previous paragraphs, however, a software task may run on any of the processors in the architecture. Furthermore, its execution may be shifted to another processor during runtime. Consequently, the task has no knowledge about which processor link to address. As a solution to this problem, we tailor conventional point-to-point schemes accordingly in order to enable the required processor-independent communication.

3.2.2. Task-based Processor-independent Communication for Virtualizable MPSoCs

In the virtualizable system depicted in Figure 3, right hand side, a unique static ID is assigned to each task in the system. In contrast to a conventional design, the communication interfaces between processors remain unconnected. As mentioned before, a task usually would address communication interfaces by their identifier, such as “send_0” or “receive_1” in Figure 3, in order to send or receive data. Instead of such an identifier, now the ID of the receiving or sending task is introduced to the send or receive command, respectively. Due to the dynamic bindings in the virtualizable MPSoC it is not known a-priori on which processors tasks will run. Executing this code on a normal MPSoC without the virtualization layer will in general cause an erratic behavior, since the task ID will not be interpreted correctly. To achieve the envisaged behavior, i.e., a correct transmission of data between tasks, the generic concept of code injection introduced in [8] needs to be modified and expanded accordingly. Please note that in this implementation we apply this technique on top of the FSL interface of Xilinx MicroBlaze soft-core processors. However, the approach is neither limited to nor dependent on this specific interface or processor type, it can easily be adapted to support other point-to-point interfaces and processor types as well.

The virtualization layer is conceived such that it scans the instruction sent to a processor from a task’s instruction memory. As soon as the virtualization layer detects the machine code of an FSL ‘send’ instruction, it modifies the instruction to be sent to the processor. Instead of the actual FSL send instruction that would cause the content

of one of the processor's registers being routed to one of the FSL interfaces, a '*store word*' command is injected into the processor code. This command lets the processor output the content of a register to the processor's data memory interface. As a memory address, the ID of the receiving task written as part of the FSL command is applied. During this process, the data and instruction memories of the sending task are temporally detached from the processor. As soon as the processor writes the register content to its data memory interface, this value is inserted in the so-called *Task Data Matrix* (TDM) as depicted in Figure 3, right hand side. For each task the TDM provides an array of registers. The output on the processor's data memory interface is written into the receiving task array at the address of the sending task.

Accordingly, if a FSL 'read' instruction is detected in the instruction stream of the receiving task, then the instruction is being modified at runtime to a '*load word*' instruction, the processor's memories are being detached, and the TDM is accessed. As soon as the value is read out from the TDM, data and instruction memories of the sending and receiving processor are re-attached. In case that no blocking occurs, an unaltered FSL transfer as well as the proposed communication via TDM features a latency of just two clock cycles.

The original virtualization approach of [8] guarantees that the execution of tasks may be interrupted at any time. A blocking write/read might theoretically prevent a task from being interrupted by virtualization, since the processor would then stall on the blocking write/read instruction, but needs to execute several lines of dedicated machine code in order to run the virtualization procedure which obviously results in a contradiction. As a solution, the injected 'store word' command for the access of the TDM is always executed. As long as the data is not fetched by the receiver, the processor remains detached from its instruction and data memory, i.e., the task execution blocks. However, since the processor has finished the execution of the 'store word' command, the processor itself is not in a blocking state, but is ready to execute subsequent instructions. Thus, the virtualization procedure, e.g., to update the processor-task bindings in the system, can still be executed even during task blocking states. A similar behavior is applied for blocking 'read' commands.

Since FSL identifiers are replaced by task IDs represented by numbers at the same position in machine code, compiling this code does not cause any warnings or errors. An obvious benefit from the proposed communication approach is that neither the compiler nor the processors need to be modified in order to support this scheme.

3.3. Invasive Programming Trigger Module

Another enhancement has to be introduced to the MPSoC architecture in order to trigger the switching process between parallel and sequential execution modes. Therefore, a dedicated *Invasive Trigger Module* (ITM) is added to the virtualization layer, cf. Figure 4. This is a tiny hardware module which communicates with tasks via a reserved slot in the TDM introduced in Section 3.2.2 and with the control processor featuring the Binding Vector Interface that manages processor-task bindings. It fetches parallelization requests from an external trigger. The ITM then requests the Binding Vector Interface to allocate processing resources according to the parallelization request. In case that there are enough free resources, the Binding Vector Interface causes the Routing Logic to perform an update of the processor-task interconnection network to route B and C code sections to allocated processors. Subsequently, the Virtualization Logic activates these tasks. Furthermore, the ITM signals to the task running section A the number of parallel instances in the B section. As soon as code section A of a task (cf. Figure 4, Task ID0) reaches a possible switching point, it may execute the generic piece of code that distributes data to the given number of parallelized sections. In future work, the ITM will be expanded to also handle parallelization requests not given by an external trigger, but being produced internally by tasks.

4. Design Methodology

After a sequential software code section is equipped with pragmas at suited positions, the automated pragma evaluation phase is resumed. As a result, several program files containing the code of the A, B, and C parts, respectively, as detailed in Section 2.2 are created. In order to smoothly set up an invasive MPSoC architecture able to execute these software portions, a dedicated tool framework called FripGa is provided as part of this work. FripGa features a task-based design flow, where software tasks are bound to an arbitrary number of processors by means of a graphic binding editor. Binding sets may be predefined by a designer or may also be created during runtime by the Binding Vector Interface based on the system state. In FripGa, both the A and the C parts as well as

Algorithm 1 Invasive Programming for Virtualizable MPSoC

Require: Sequential algorithm t
Ensure: MPSoC being able to toggle parallel and sequential instances of t
1: Insert pragma in t at a section suited for parallelization
2: Evaluate pragma to gain fractions t_a, t_{b_i}, t_c of t
3: Identify a switching point SP in t for the change of execution mode
4: Merge automatically generated code for dependency resolution of t_a into t at SP
5: Define task IDs for t, t_{b_i}, t_c and insert IDs in 'send'/'receive' commands
6: Set up MPSoC by adding t, t_{b_i}, and t_c as task elements in FripGa design framework
7: Define maximum number of allocable processors
8: **for all** representations of t, either sequential or parallel, **do**
9: Define processor-task binding
10: Define trigger activating the current mode
11: **end for**
12: Generate configuration bitstream

all the B parts are added to the related so called sw project. The number of added B parts is the upper bound for the parallelization degree during runtime. At runtime an arbitrary number of B parts may be active up to this number. As FripGa is compatible to FPGA vendor design flows, it is possible to simulate the entire design and to generate configuration bitstreams for, e.g., Xilinx FPGAs. When triggering bitstream generation, the virtualization layer as depicted in Figure 4, required processor instances and various default design IP cores, such as clock generators, are automatically inserted into the design. The resulting hardware is able to switch between the sequential and parallel execution modes by means of the techniques detailed in Sections 2.2 and 3. All requirements postulated in [4] are met at this point: (a) the pragma evaluation automatically inserts communication commands supported by the target architecture (b) the architecture allows for a dynamic binding of programs to processing units (c) programs featuring pragmas may be compiled and the entire system may be simulated due to the compatibility of FripGa with design tools such as ModelSim. Thus, invasive designs may now be generated following the methodology denoted in Algorithm 1.

Up to now, some manual interaction is still necessary regarding the placement of pragmas. However, existing code analysis tools may be exploited in order to identify suited places for automatic pragma placement. Merging of the code fraction t_a into t as outlined in Algorithm 1, step 4, is currently being automated. As a consequence, pragma evaluation will automatically insert the fractions t_{b_i}, and t_c into t featuring the code for dependency resolution of t_a at the switching point.

5. Application Example

For demonstration purposes, an image downscaling algorithm based on a digital filter with finite impulse response was coded in C as sequential software. We have chosen this application, since it offers several ways to easily parallelize the code. In this example, we run concurrently the process of computing the color information of pixels of a line of the resulting downscaled picture, cf. Figure 5. Other parallelization points, such as parallel filtering for each color channel of each pixel, might have been chosen as well. We generated two parallel variants by setting the degree of parallelization to 2 and 4, respectively. Thus, in each of these concurrently operating variants, 2, or 4 pixels in a line of the resulting image are computed in parallel. The pure sequential process of generating a pixel of the resulting image on a static single-core system takes about 9,000 clock cycles. At first, this is taken into comparison to a static multi-processor solution as depicted in Figure 6. E.g., for the implementation, where two processors are fed with current pixel information pixels of the output image are generated at an average of 5,900 clock cycles. The maximal theoretical speed-up of factor 2 is not reached due to sequential overhead, such as defining the window tab of the original line corresponding to the pixel of the output line. The application example surely can be further optimized, but this is not in the scope of this contribution.

As a matter of fact. for higher levels of parallelization, the speed-up factor lowers, because the time needed for the sequential transfer of data to the parallel sections increases with an increasing number of parallel sections. Therefore, parallel sections have to wait longer until the data needed for computation is received.

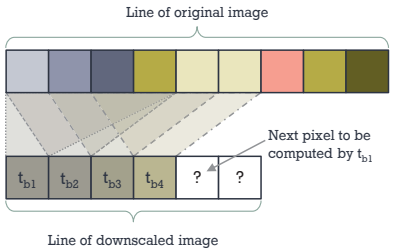

Fig. 5. Parallel Filtering of 4 Pixels of a downscaled Image.

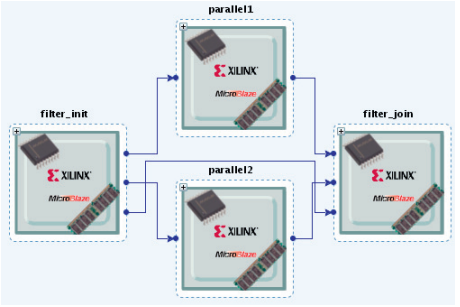

Fig. 6. Static Multi-Processor System represented in FripGa Tool.

Table 1. Comparing Sequential, Static Parallel, and Invasive Parallel Solutions.

	Static Solutions			Invasive Solutions		
Timing values given in clock cycles	Seq.	2x Par.	4x Par.	Seq.	2x Par.	4x Par.
Time per Pixel[1]	9,076	5,900	4,253	9,082	5,973	4,326
Transferring Data to a parallel Instance[2]	-	5	5	-	56	56
Specific Timing Overhead for invasive Features						
Interpreting ITM Status in Section A				6	6	6
Switching Execution Mode[3]				155	155	155

We now compare the sequential and static parallel solutions to our virtualization architecture by executing the same software code with the adaptions added for the invasive scheme. Please note that the main contribution of the proposed solution is not to provide a higher speed-up compared to static multi-processor solutions, but a significant yield in terms of the range of execution dynamics. With the proposed solution any design may mutate during runtime from a purely sequential to an adjustable parallel implementation featuring various parallelization degrees, which is of utmost importance when considering adaptable power consumption and fault tolerance issues. However, the question is whether the prevenient overhead for invasion handling, data forking, and re-accumulation within the virtualizable architecture is still feasible in comparison to static solutions.

One may easily realize from the timing results summarized in Table 1 that the invasive features result in a slight overhead for transferring data dependencies between tasks. Timing in the parallel stages is merely affected by longer data transfers caused by accessing the TDM and generic wrappers for 'send'/'receive' commands generated during pragma evaluation. While this scheme is well-suited for data-flow heavy applications, scenarios that rely on a high number of memory accesses may cause a considerable overhead in terms of additional send/receive events. For these scenarios, globally addressable memories for invasive architectures, such as those examined in [13], might be better suited at the expense of some disadvantages of shared-memories as detailed in Section 3.2.1.

For the sequential solution executed on the architecture, a negligible overhead is caused by checking the status of the ITM in each turn, i.e., before each pixel computation. In case of a parallelization request a virtualization procedure is triggered, processor resources are allocated, and the parallel task sections are routed to these processors. As soon as the task featuring section A receives the information from the ITM that the system reconfiguration is applied, it may start distributing data to the parallel sections. Setting up a new processor-task binding takes 155 clock cycles only. This virtualization approach clearly outperforms conventional thread switches as found in common embedded kernels such as the Xilkernel from Xilinx. For Xilkernel, a task switch takes about 1,350 clock cycles. The Virtualization Layer for an eight processor MPSoC occupies about just 2.8 % of available registers and about 10.7 % of lookup tables of the employed Virtex-6 LX240T FPGA.

Another interesting application addresses a system in which Invasive Programming enables the usage of en-

[1] Time intervals at which filtered pixel data is written to the output stage.
[2] The filter's tab size is 4. For each computation, current phase as well as 24-bit color information of 4 pixels of the original image are needed.
[3] This triggers a reconfiguration of the processor-task network and virtualization procedures which allocate/deallocate processors.

ergy awareness features. Mobile devices with a rather limited supply of energy may serve here as a suited field of application. In normal operation, an application is executed in parallel execution mode to gain maximum performance. In case of a shortcoming in energy supply, the degree of parallelization may dynamically be reduced. In consequence, the Virtualization Layer deallocates the soft-core processor resources now being unused. Additionally, these processing units as well as their components are deactivated by disabling their clock inputs. First measurements have shown that the deactivation of a processor in the array may reduce the energy consumption by about 10 %. The effectiveness of these energy awareness features is currently being evaluated by a series of measurements conducted in different operating modes.

6. Conclusion

In this paper, a methodology for an invasive programming paradigm on a virtualizable MPSoC was presented. It combines a designer-guided code parallelization approach for embedded multi-processor systems with a generic MPSoC architecture featuring dynamic task migration and processor (de-)allocation at runtime. As one of the novel contributions, the virtualizable MPSoC architecture was enhanced by a processor-independent task-based communication scheme which now allows transferring data between tasks without needing any information on which processor tasks are currently being executed. This scheme enables the resolution of dynamic data dependencies between tasks during runtime despite a transparent task migration. The virtualization layer of the MPSoC was expanded to feature the dynamic switch between sequential and parallel execution modes. Parallelized software tasks may then invade unused processors in the processor array. In case of failing processor resources or shortcomings in energy supply, the degree of parallelization may dynamically be decreased or the sequential execution mode may be triggered. A design tool set to efficiently handle software tasks, processors, and processor-task binding sets for sequential or parallel execution scenarios was introduced. By means of the proposed methodology and the related tool set, a smooth and comprehensive creation of MPSoCs featuring an invasive programming paradigm is being supported.

In future, the rivaling behavior of several tasks being able to invade unused processors will be considered. Given external factors, such as variations in energy supply or defective processors which lead to the deactivation of parts of the processor array, the dynamic creation of suited processor-task bindings as well as the automated triggering of the invading behavior is already in focus of our on-going research.

References

[1] L. Dagum, R. Menon, OpenMP: An Industry Standard API for Shared-Memory Programming, Computational Science Engineering, IEEE 5 (1) (1998) 46–55. doi:10.1109/99.660313.

[2] A. Papakonstantinou, et al., FCUDA: Enabling efficient Compilation of CUDA Kernels onto FPGAs, in: Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on, 2009, pp. 35 –42. doi:10.1109/SASP.2009.5226333.

[3] T. e. a. Czajkowski, From OpenCL to High-Performance Hardware on FPGAS, in: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, 2012, pp. 531 –534.

[4] J. Teich, Invasive Algorithms and Architectures, in: it - Information Technology, Vol. 50, No. 5, 2008, pp. 300–310.

[5] A. Biedermann, M. Zöllner, S. A. Huss, Automatic Code Parallelization and Architecture Generation for Embedded MPSoC, in: ACM 5th Workshop on Mapping of Applications to MPSoCs (Map2MPSoC/SCOPES 2012), 2012.

[6] Vector Fabrics, Pareon, `www.vectorfabrics.com/products/`.

[7] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, J. Weidendorfer, Invasive Computing with iOMP, in: Specification and Design Languages (FDL), 2012 Forum on, 2012, pp. 225 –231.

[8] A. Biedermann, M. Stoettinger, L. Chen, S. A. Huss, Secure Virtualization within a Multi-Processor Soft-core System-on-Chip Architecture, in: The 7th International Symposium on Applied Reconfigurable Computing, Belfast, UK, 2011.

[9] H. Simmler, L. Levinson, R. Männer, Multitasking on FPGA Coprocessors, in: R. W. Hartenstein, H. Grünbacher (Eds.), FPL 2000, Vol. 1896 of LNCS, Springer, 2000, pp. 121–130.

[10] P. Varanasi, G. Heiser, Hardware-supported Virtualization on ARM, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys '11, ACM, New York, NY, USA, 2011, pp. 11:1–11:5.

[11] A. Molnos, et al., Decoupled inter- and intra-application Scheduling for composable and robust embedded MPSoC platforms, in: ACM 5th Workshop on Mapping of Applications to MPSoCs (Map2MPSoC/SCOPES 2012), 2012.

[12] A. Biedermann, S. A. Huss, Hardware Virtualization-driven Software Task Switching in Reconfigurable Multi-Processor System-on-Chip Architectures, in: ACM 5th Workshop on Mapping of Applications to MPSoCs (Map2MPSoC/SCOPES 2012), 2012.

[13] J. Teich, A. Weichslgartner, B. Oechslein, W. Schroder-Preikschat, Invasive Computing - Concepts and Overheads, in: Specification and Design Languages (FDL), 2012 Forum on, 2012, pp. 217 –224.